表示的意味論

AUTHOR(S):

久木田, 水生

CITATION:

久木田, 水生. 表示的意味論. 哲学論叢 2008, 35: 189-198

ISSUE DATE:

2008

URL:

http://hdl.handle.net/2433/96270

表示的意味論

久木田水生

1. 序

表示的意味論*denotational sematics* はプログラミング言語 —— 特に関数型言語 —— に対する意味論の一種である。「表示的」という言葉は、任意の式に対してある数学的構造の中の一つの対象を割り当てるという特徴に由来している。プログラミング言語に対する主要な意味論としては、他に**操作的意味論***operational semantics* がある。操作的意味論では、ある式を（仮想的）有限状態機械(1) に対する状態遷移命令とみなす (cf. Stoy, 1977, chap. 2)、あるいは特定のインタプリタの仕様を考え、与えられた式をそのインタプリタが変換して得られる正規形の式(2) をもとの式の値と見なす (cf. 横内, 1994, sec. 4.2)。

実際のプログラムはコンピュータに何らかの動作をさせるものであるから、プログラミング言語の意味論としては操作的意味論の方がより自然に思われる。一方で、表示的意味論には、プログラムの抽象的な構造に焦点を当て、その数学的側面を研究することを可能にするという利点がある。実際のところは、表示的意味論の適切さは、操作的意味論との整合性によって計られる —— あるいはそもそも、表示的意味論はプログラムを操作的に捉える直観に基いて構築される —— ため、両者の違いはプログラムに対する観点の違いであるということも出来る (cf. 横内, 1994, 第 4 章)。しかしながら表示的意味論は、純粋に数学的な研究分野であるに留まらず、言語やコンパイラの設計、プログラムの正しさの検証といった実用の場面においても重要な役割を果たしている。さらに Allison (1986) は表示的意味論が「論理学、哲学、言語学におけるモデル理論を含む、幅広い運動の一部」（p. 3）であると述べる。横内 (1994) もまた表示的意味論の考え方が「プログラム言語に限らず、すべての形式的言語の意味論に応用できる」（p. 6）と述べる。

確かに、ここ半世紀の間、表示的意味論とその基礎になる領域理論は、計算論、計算機科学、論理学などにおいて大きな発展を遂げてきた。しかしこれらの分野での発展と比較したとき、表示的意味論が哲学に対しても相応のインパクトを与えてきたとはまだ言えない。その理由の一つは、プログラミング言語は非常に限定された目的 —— コンピュータに特定の仕事をさせるように命じるという目的 —— に特化された言語であるため、その意味論が言語や数学一般に応用できるように思われない、ということがあるだろう。

しかしながら、論理学、数学、計算機科学、人工知能、認知科学、言語学などの分野の結びつきがますます緊密になっている現状をみれば、プログラミング言語の意味論が、数学の哲学や言語哲学においても有用な洞察をもたらす可能性は高い。そこで本稿では、表示的意味論の中心的なアイディアを紹介し、その上で表示的意味論が哲学において持ちうる重要性について考察したい。

本稿では、集合論と型無しラムダ計算についての基本的な知識は前提されるが、特定のプログラミング言語についての知識は前提されない(3)。

2. 表示的意味論の特徴

表示的意味論とは、ある形式的言語の表現に対して、ある数学的構造における対象を割り当てることによって、その表現の意味を決定する理論である。この数学的構造は**領域***domain* と呼ばれる。また領域の対象の言語に対する割り当ては**表示関数***denotational function*、あるいは**意味関数***meaning function* と呼ばれる。従って表示的意味論は、(1) 文法、(2) 領域、(3) 表示関数からなる。例えば古典命題論理における真理値付値も表示的意味論の一つである。その場合、領域は真理値の集合 $\{0, 1\}$ になる。

プログラミング言語に対する意味論は論理や算術の形式言語と大きく異なる点を持っている。ここではそのいくつかを考察しよう。以下ではプログラミング言語に対する表示的意味論を単に表示的意味論と呼ぶ。

コンピュータは記号を操作することによってデータの処理・加工を行う。従ってコンピュータの内部には、静的なデータと、動的なプロセスが存在している。データは何らかの情報を担い、プロセスはある情報を担うデータを他の情報を担うデータへと変換する。プログラミングの目的は、これらのデータとプロセスを制御することである。従ってプログラミング言語に現れる式は何らかのデータ、もしくはデータを変換するプロセスを表わしていると考えることが出来る。

多くのプログラミング言語において、最も基本的なデータは整数、文字列、真理値（**true**、**false**）、未定義値（**undef**）などである。最も基本的なプロセスは整数上の演算、文字列の操作、真理値の計算などである。さらにプロセスの流れを制御する命令（条件分岐や繰り返し）、変数にデータを割り当てる命令、関数名にプロセスを割り当てる命令が含まれる。プログラミングはこれらの要素を適切に組み合わせることによって作られる。

表示的意味論の基本的なテーゼは、データの領域を完備順序集合とし、プロセスの領域を完備順序集合間の連続関数の集合とすることである。ここでいくつか必要な定義を述べておこう。以下では半順序 partial order を単に順序と呼ぶ。

2.1 CPO と連続関数

定義 2.1 (有向集合) 順序集合 $(A, \preceq)$ が**有向** *directed* であるのは、任意の $x, y \in A$ に対して、ある $z \in A$ が存在し、$x \preceq z$、$y \preceq z$ が成り立つときである。

定義 2.2 (完備順序集合) 順序集合 D が**完備** *complete* であるのは、D が最小元を持ち、かつ D の任意の有向部分集合 A に対して、その上限が D に含まれるときである。

以後、有向集合 A の上限を $\vee A$ と表記する。また完備順序集合を CPO（complete partial ordered set の略）と呼ぶ。

定義 2.3 (単調関数) $(D, \leq), (D', \leq')$ は順序集合であるとする。関数 $f : D \to D'$ が**単調** *monotonic* であるのは、任意の $x, y \in D$ に対して、$x \leq y$ ならば $f(x) \leq' f(y)$ が成り立つときである。

関数 $f : D \to D'$ と、$A \subseteq D$ に対して、A の f による順像、すなわち $\{f(x) \in D' \mid x \in A\}$ を $f[A]$ と表記する。次の事実が成り立つことは容易に理解される：任意の順序集合 D, D' と単調関数 $f : D \to D'$ と D の任意の有向部分集合 A に対して、$f[A]$ は D' の有向部分集合である。

定義 2.4 (連続関数) CPO、D, D' に対して、関数 $f : D \to D'$ が**連続** *continuous* であるのは、任意の有向集合 $A \subseteq D$ に対して、$f(\vee A) = \vee f[A]$ が成り立つときである。

$(D, \leq), (D', \leq')$ は CPO とする。このとき D から D' への連続関数全体の集合を $[D \to D']$ と表わす。また $[D \to D']$ 上の順序関係 $\leq$ を次のように定める。任意の $f, g \in [D \to D']$ に対して、

$$f \leq g \iff \forall x \in D(f(x) \leq' g(x)).$$

以後、$[D \to D']$ を順序集合として扱うときには、断りがなければこのように順序が定義されているものとする。

命題 2.5 D, D' は CPO とする。このとき $[D \to D']$ は CPO である。

証明 横内 (1994)、定理 3.2.4 の証明を参照せよ。 □

定義 2.6 (部分関数) A, B は集合とする。$f \subseteq A \times B$ に対して、ある関数 $f' : A \to B$ が存在し、$f \subseteq f'$ が成り立つとき[4]、すなわち任意の $a \in A, b \in B$ に対して $\langle a, b\rangle \in f$ ならば $f'(a) = b$ が成り立つとき、f を A から B への**部分関数** *partial function* と呼び、$f : A \rightharpoonup B$ と表わす。従ってもちろんすべての関数は部分関数である。

$a \in A$ に対して、ある b が存在して $\langle a, b\rangle \in f$ が成り立つとき、$f(a)$ は**定義されている** *defined* といわれ、この b を $f(a)$ と表わす（このような b は一意である）。一方、$\langle a, b\rangle \in f$ なる b が存在しないとき、$f(a)$ は**未定義である** *undefined* といわれる。

部分関数 $f : A \rightharpoonup B$ と $g : B \rightharpoonup C$ に対して

$$\{\langle a, c\rangle \in A \times C \mid \exists b \in B(\langle a, b\rangle \in f \ \&\ \langle b, c\rangle \in g)\}$$

は A から C への部分関数になる。これを f と g の**合成** *composition* と呼び、$g \circ f : A \rightharpoonup C$ と表わす。

CPO 間の部分関数についても、通常の関数と同様に連続性を定義することが出来る。$(D, \leq)$ から $(D', \leq')$ への連続部分関数全体の集合を $[D \rightharpoonup D']$ と表わす。$[D \rightharpoonup D']$ に

$$f \leq g \iff \forall d \in D(f(d) \text{が未定義であるか、または } f(d) \leq' g(d))$$

によって順序 $\leq$ を導入すると $([D \rightharpoonup D'], \leq)$ も CPO になる。

定義 2.7 (近似) $f, g : D \rightharpoonup D'$ は連続部分関数とする。任意の $d \in D$ に対して、$g(d)$ が未定義であるか $g(d) = f(d)$ のどちらかであるとき、g は f の**近似** *approximation* であるといわれ、$g \sqsubseteq f$ と書かれる。特に $g(d)$ が未定義にならないような $d \in D$ すべての集合が有限であるとき、g は f の**有限近似** *finite approximation* であるといわれ、$g \sqsubseteq_{\text{fin}} f$ と書かれる。

定義 2.8 (不動点) 関数 $f : A \to A$ に対して $f(a) = a$ を満たす $a \in A$ を、f の**不動点** *fixed point* という。

任意の関数（部分関数）$f : A \to A$ と $n \geq 0$ に対して、f^n は、

$$f^n = \begin{cases} id_A & (n = 0 \text{ のとき}) \\ f \circ f^{n-1} & (n > 0 \text{ のとき}) \end{cases}$$

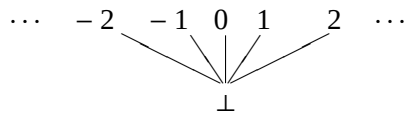
によって定義される A 上の関数を表わすものとする。ただし id_A は A 上の恒等関数。

命題 2.9 D は CPO、$\perp$ は D の最小元、$f : D \to D$ は連続であるとする。このとき $\vee\{f^n(\perp) \mid n \geq 0\}$ は f の最小不動点である。かつ $\text{fix} : [D \to D] \to D$ によって任意の D 上の連続部分関数からその最小不動点への関数を表わすとすると、fix は連続関数である。

証明 高橋 (2003)、定理 3.2.17 の証明を参照せよ。 □

2.2 データとプロセスの領域

表示的意味論においては、領域は複数の基本的な CPO、およびそれらの間の（部分）連続関数の集合によって構成される。領域を順序集合とするのは、プログラミングの扱う対象は何らかのデータであり、データには情報の多少があると考えられるからである。ただし整数のようなデータの間には情報の多少はない。従って整数全体のなす CPO は通常の大小関係によって順序付けられるわけではない。その代わりに整数の集合には未定義の値を表わす対象 $\perp$ が付け加えられ、これが最小元であるとされる。この集合を $\mathbf{Z}_\perp$ とすると、$\mathbf{Z}_\perp$ は次のような順序型を持つ。



したがって $\mathbf{Z}_\perp$ 上の関数の順序は

$$f \sqsubseteq g \iff \text{任意の } x \in \mathbf{Z} \text{ に対して } f(x) = \perp \text{ または } f(x) = g(x)$$

によって定められる。$\perp$ は未定義値を表わすと考えれば、この順序関係は上で定義された近似の関係に他ならない。

整数を引数として受け取り、整数を値として返す手続きを表わすプログラムの表示的意味の領域は $\mathbf{Z}_\perp$ 上の関数をもとにして考えられるが、しかしもちろん整数上のすべての（部分）関数がプログラムで書けるわけではないので、求める領域は $\mathbf{Z}_\perp$ 上の関数の部分領域になる。その領域が完備でなければならない理由を、以下で例を挙げて説明しよう。

次のように定義される関数 fact を考えよう。

$$\mathrm{fact}(x) ::= \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * \mathrm{fact}(x-1)$$

この関数は任意の非負整数 x に対して、$f(x) = x!$ を満たす、つまり階乗の計算をする関数である。この定義は再帰的に行われているため、その部分式の意味から全体の意味を決定することは出来ない。$\mathrm{fact}(x)$ を定義に従って展開していくと、

$$\begin{aligned}\mathrm{fact}(x) \equiv\ & \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * \\ & (\textbf{if } (x-1) = 0 \textbf{ then } 1 \textbf{ else } (x-1) * \\ & \quad (\textbf{if } (x-2) = 0 \textbf{ then } 1 \textbf{ else } (x-2) * \\ & \quad\quad (\textbf{if } (x-3) = 0 \textbf{ then } 1 \textbf{ else } (x-3) * \cdots\end{aligned}$$

となって、無限に長い式が出来てしまう。しかし私たちは任意の $n \geq 0$ に対して、

$$\mathrm{fact}_0(x) ::= \textbf{undef}$$

$$\mathrm{fact}_1(x) ::= \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else undef}$$

$$\begin{aligned}\mathrm{fact}_2(x) ::= & \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * \\ & (\textbf{if } (x-1) = 0 \textbf{ then } 1 \textbf{ else undef})\end{aligned}$$

$$\begin{aligned}\mathrm{fact}_3(x) ::= & \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * \\ & (\textbf{if } (x-1) = 0 \textbf{ then } 1 \textbf{ else } (x-1) * \\ & \quad (\textbf{if } (x-2) = 0 \textbf{ then } 1 \textbf{ else undef}))\end{aligned}$$

という仕方で関数 fact_n が定義できる。表示的意味論ではこれらの意味は次のようにして定められる。まず $\Phi : [\mathbf{N} \rightharpoonup \mathbf{N}] \to [\mathbf{N} \rightharpoonup \mathbf{N}]$ を次のように定めよう：

$$\Phi = \lambda f : [\mathbf{N} \rightharpoonup \mathbf{N}].\lambda x : \mathbf{N}.(\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x * f(x-1))$$

ただし λ はメタのラムダ抽象を表わす。このとき Φ は連続である。$\emptyset : \mathbf{N} \rightharpoonup \mathbf{N}$ はすべての $n \in \mathbf{N}$ に対して $\emptyset(n)$ が未定義であるような部分関数であるとする。$\emptyset$ は連続であるから、$\emptyset$ は $[D \rightharpoonup D]$ の最小元になっている。このとき任意の $n \geq 0$ に対して

$$\Phi^n(\emptyset)(x) = \begin{cases} x! & (x < n \text{ のとき}) \\ \text{未定義} & (x \geq n \text{ のとき}) \end{cases}$$

が成り立つ。従って fact_n の表示的意味として、$\Phi^n(\emptyset)$ を指定することが自然である。このとき $\Phi^0(\emptyset) \sqsubseteq_{\mathrm{fin}} \Phi^1(\emptyset) \sqsubseteq_{\mathrm{fin}} \ldots$ かつ $\mathrm{fact} = \vee\{\Phi^n(\emptyset) \mid n \geq 0\}$ が成り立つ。また命題 2.9 より $\mathrm{fact} : \mathbf{N} \to \mathbf{N}$ は Φ に関する最小不動点である。そこで fact の表示的意味としては $\vee\{\Phi^n(\emptyset) \mid n \geq 0\}$ を指定することが出来る。このように領域を CPO および CPO 間の連続（部分）関数とすることによって、上記のような再帰的に定義される関数に対して、連続関数 Φ の最小不動点として、その表示的意味を特定することが出来る。

以上が CPO を領域とする表示的意味論の基本的なアイディアである。再帰的な仕方でプログラムされる関数は様々あり、それらに統一的な仕方で表示的意味を指定することが可能か、ということは問題であるが、連続関数についていえばその不動点を計算する手続きがあれば、その表示的意味が取れる(5)。

CPO での上限として再帰関数に表示的意味を与えるということの意味を考えてみよう。コンピュータは有限のプログラムから終わりのないプロセスを展開することが出来る。これはプログラミング言語が手続きの再帰的呼び出し（繰り返し）を可能にする文法を持っているからである。このようなプログラムを実行するときコンピュータは実質的には同じ手続きを繰り返し実行している。一回一回の実行の度にその手続きは特定の結果を生み出し、その結果が再び同じ手続きに対して入力として与えられる。このように無限に展開しながらその都度値を返し続けるプロセスを生み出すプログラムに対して、確定した意味を与えることは困難に思われる。しかしながら無限に生み出される値が、無秩序に散らばるのではなく、決まった点に方向付けられてることが保証されている場合、それらの値の列が向かっていく先にあるものを、そのプロセスを生み出すプログラムが持つ値であると見なすことは自然である。例えば数列 $\{1/n\}_{n>0}$ の表わす値は何かといわれれば、私たちは 0 と答えるだろう。なぜならばこの数列はどこまでも 0 に近づいていく（0 に収束する(6)）ことが証明されるからである。この意味で 0 はこの数列にとって特別な値であり、かつ 0 以外のいかなる点も特別視される理由を持たない。

ただしこのように数列が収束していく先を、その数列が表わす値であると考えることが出来るのは、その収束先に対象（極限）が存在していることが保証されている場合に限る。ここで領域が CPO であるという想定が意味を持って来る。順序集合における有向集合は、直観的に言えば収束する数列と類比的なものである。そして D が CPO であるならば、任意の有向部分集合 $A \subseteq D$ に対して極限（上限）$\vee A$ が存在することが保証されている。それゆ

えに私たちは fact の値として $\vee\{\text{fact}_n \mid n \geq 0\}$ を指定することが出来るのである。

このようにあるプログラムの表示的意味を、その有限近似の上限とする方法は、$\sqrt{2}$ などの無理数を計算するプログラムに対しても適用できる。無理数は有限の文字列や数列で表現することが出来ない。有限のメモリと有限の計算時間しか持たないコンピュータにとって、無理数は常に近似によって扱われる。しかし私たちはどこまでも $\sqrt{2}$ に近い有理数を求める手続きを知っている。より正確に言えば、任意の有理数 q に対して $|\sqrt{2} - q'| < |\sqrt{2} - q|$ を満たす q' を求める手続きを知っている。その手続きの一つは、$f(x) = 2/x$ という関数の不動点を求める手続きとして、例えば次のように書くことが出来る (cf. Abelson & Sussman, 1996, sec. 1.3.3)。

$$f(x) ::= 2/x$$

$$\text{sqrt}(f, x) ::= \text{sqrt}(f, (f(x) + x)/2)$$

もちろんこのように書かれたプログラムは終了することはないが、例えば f を呼び出す回数を制限することによって、近似的なプログラムを書くことが出来る。f を n 回呼び出し、その時点での値を返すプログラムを sqrt^n と呼ぶことにすると、sqrt^n は n が大きければ大きいほど $\sqrt{2}$ に近い値を出すことになる。従って、各 n に対して sqrt^n の表示的意味を $[\![\text{sqrt}^n]\!]$ によって表わすことにすれば sqrt の表示的意味 $[\![\text{sqrt}]\!]$ は

$$[\![\text{sqrt}]\!] = \vee\{[\![\text{sqrt}^n]\!] \mid n \geq 0\}$$

によって定義することが出来るだろう。

3. 表示的意味論の哲学における重要性

表示的意味論は、プログラムを CPO の中の対象に対応させることによって、動的なコンピュータ・プロセスを静的な数学的対象として扱うことを可能にしてきた。このことによって、CPO という一般的な数学的構造について知られている多くの事実が、プログラミングに応用できるようになった。しかし表示的意味論はこのような実際的な問題に対してのみならず、数学や論理学の哲学における諸問題に対してもまた重要なインパクトを持つ可能性がある。この節では表示的意味論が哲学において持ちうる重要性について考察しよう。

従来、数学の哲学においては、命題の意味と真理とが常にその中心的なテーマとして扱われてきた。そして一般的には、命題とは何らかの対象の状態を表現するものであり、命題が真であるのはそれが描写している状態が実際に成り立っているときである、と考えられている。従って命題は常に何らかのモデルと関係付けられて解釈されるのである。このような言語観、真理観をモデル論的解釈と呼ぶことにしよう。モデル論的解釈は哲学者の間では非常に根強いものであり、数学の哲学における議論においてしばしば何の疑問も持

たれないままに前提されている。それゆえ、コンピュータに特定の仕事をさせる命令を書くという目的に特化されたプログラミング言語の意味論が、数学の哲学においてこれまでほとんど注目されて来なかったのも不思議なことではない。

しかしながら実際はプログラミング言語と数学や論理学の言語には密接な関係がある。例えば Lisp は型付ラムダ計算の直接の応用である一方、型付ラムダ計算の式は自然演繹での証明図に、式の型はその証明図によって証明される論理式に対応する (cf. Troelstra & Schwichtenberg, 2000, sec. 2.2 and chap. 6)。また Prolog のプログラムは一階述語論理の言語そのものであり、Prolog インタプリタの動作は、導出原理を用いた節形論理における証明の過程である (cf. Kowalski, 1979; Robinson, 1965)。数学や論理学の式を表わすために発達した言語が、プログラミング言語に応用される際には、「これこれの計算（推論）を遂行せよ」という命令を表わすものとして解釈される、ということは注目に値する。というのも、数学と論理学の言語に対する同様の解釈の転換が、プログラミング言語に応用する際のみならず、数学や論理学の内部でも起こっているからである。

19 世紀後半から 20 世紀初頭にかけて起こった数学の基礎付けの運動の中で、数学の言語を従来とは異なる仕方で解釈することが直観主義者たちによって示唆されてきた。彼らは数学の命題を「遂行されるべき課題」として、また命題の証明を「課題を遂行する手続き」として解釈することを提案する。彼らに従えば、ある命題 *P* について、「*P* が真である」という主張は、「*P* が遂行されうる」あるいは「*P* を遂行する手続きが構成されている」という主張として理解される。数学の言語に対するこの種の解釈は**ブラウワー＝ハイティング＝コルモゴロフ解釈**（BHK 解釈）と総称される。BHK 解釈の一つのヴァージョンでは、数学的命題の証明とは、求める条件を満たす数学的対象の構成法の指示、すなわちプログラムであるとされる。この解釈はカリーとハワードによって示された、型-命題-集合の同型性（**カリー＝ハワード同型**）によって根拠を与えられる (cf. Martin-Löf, 1984, pp. 5f)。彼らは、ある対象がある集合の要素であるということと、あるラムダ項がある型を持つということと、ある構成がある命題を証明するということの間には同型性が成り立つということを示したのである。数学の言語に対するこのような解釈を証明論的解釈と呼ぶことにしよう。

モデル論的解釈をとる人々にとって、数学的対象は人間の活動とは独立の存在者である。一方、証明論的解釈をとる人々にとって、数学的対象は数学的言語による構成と独立に存在するものではない。ある数学的対象が存在するということは、その対象が合法的な手続きによって構成されるということに他ならないからである。20 世紀前半の構成主義数学の発展は、コンピュータ・サイエンスの発展とともに、数学の命題や証明、さらには数学的対象に対するこのような新しい解釈を数学者共同体の中の一角に定着させることになった。

1960 年代、ダナ・スコットは形式的な記号操作の体系であるラムダ計算に対する CPO モデルを発見した。さらに型付ラムダ計算に対しては CPO モデルの他に、圏論においてデカルト閉圏 cartesian closed category によるモデルを与える試みが行われた[(7)]。70 年代から 80

年代にかけてマーティン-レーフは数学の基礎となる体系として、「直観主義的型理論」と呼ばれる述定的な predicative 従属型理論 dependent type theory を発展させた。そして従属型理論に対しては局所デカルト閉圏 locally cartesian closed category によるモデルが与えられた (cf. Jacobs, 1999, chap. 10)。またラムダ計算に対する位相空間による表現についての研究も行われている (cf. Awodey, 2000)。

一見するとこれらの成果は再び証明論的解釈からモデル論的解釈への回帰を促すもののように思われるかもしれない。しかしこれらの成果を従来のモデル論的観点から眺めることは適当ではない。というのも前節で見たように、表示的意味論におけるモデル（領域）は決して言語と独立に、いわば天下り的に与えられるようなものではないからである。私たちが CPO の中に上限を見出すのは、いかなる有限近似も超えて、より正確な値を計算するプロセスを生み出すようなプログラムが実際にあるからである。表示的意味論が提示するモデルはプロセスを生み出す手続きの抽象化であり、プロセスの動的側面を捨象したものである。従って表示的意味論はむしろモデル論的解釈と証明論的解釈の間の境界をまたぎ、そしてその境界をぼかす。言語と世界というような単純な二分法はここでは意味を成さない。

ラムダ計算、型理論、そしてそのモデルの研究に従事している数学者たちの実践は、従来の数学の哲学が好んで使ってきた「プラトニズム」、「概念主義」、「唯名論」というステレオタイプのどれにも当てはまらない。上記の数学者たちは、数学の言語やプログラミング言語が、プラトン的なイデアの世界を記述しているとも、精神の中に存在する観念を表わしているとも、まして無意味な記号の列であるとも考えていない。少なくとも計算論や証明論の文脈において、言明は世界の状態を表わし、その状態が成立しているときに当の言明が真になるという解釈は完全に的外れである。表示的意味論は少なくとも私たちに、このような議論から袂を分かって向かうべき道を示してくれている。

註

(1) 有限状態機械とは、有限の内部状態を持ち、ある記号を受け取った時に、決まった仕方で内部状態を変化させる抽象的な機械である。例えば内部状態の集合 $Q = \{q_0, q_1\}$ と、文字の集合 $A = \{a_0, a_1\}$ に対して、関数 $\delta : Q \times A \to Q$ が次のように定義されているとしよう。

$$\delta(q_i, a_j) = \begin{cases} q_i & (i = j \text{ のとき}) \\ q_{1-i} & (\text{それ以外}) \end{cases}$$

ただし $i, j = 0, 1$。このとき $\langle Q, A, \delta \rangle$ は一つの有限状態機械を特定する。有限状態機械への入力は文字列によってなされ、その文字列を一文字ずつ読んでいき、最後の文字の読み取りが終わったときの内部状態がその入力に対する出力になる。チューリング・マシンも有限状態機械の一種である。

(2) それ以上単純な形に還元することが出来ない式を正規形の式という。

(3) 型無しラムダ計算については高橋 (2003) が好適の入門書である。

(4) 関数 $f : A \to B$ は関係の一種、すなわち $A \times B$ の部分集合として考えられることに注意せよ。

(5) 型無しラムダ計算であれば例えばカリーの不動点コンビネータ $\lambda y.y((\lambda x.y(xx))(\lambda x.y(xx))$ によって、任意の関数の不動点を計算することが出来る。

(6) ある有理数列 $\{a_n\}_{n>0}$ が x に収束するというのは、任意の自然数 k に対してある自然数 N が存在して、すべての自然数 $m \geq N$ に対して、$|x - a_m| < 1/k$ が成り立つということである。

(7) 圏論についての説明は本稿では割愛する。圏論の基礎については例えば Lawvere & Schanuel (1997); Awodey (2006) などを、圏論と型理論については Lambek & Scott (1986), part I, section 11 などを参照せよ。

文献

Abelson, H. & Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*, Massachusetts: The MIT Press, 2nd edition.

Allison, L. (1986). *A Practical Introduction to Denotational Semantics*, *23* of Cambridge Computer Science Texts, Cambridge: Cambridge University Press.

Awodey, S. (2000). 'Topological representation of the λ-calculus,' *Mathematical Structure in Computer Science*, *10*, 81–96.

——— (2006). *Category Theory*, New York: Oxford University Press.

Jacobs, B. (1999). *Categorical Logic and Type Theory*, *141* of Studies in Logic and the Foundations of Mathematics, Amsterdam: Elsevier.

Kowalski, R. (1979). *Logic for Problem Solving*, Amsterdam: Elsevier North Holland, Inc.

Lambek, J. & Scott, P. J. (1986). *Higher Order Categorical Logic*, Cambridge: Cambridge University Press.

Lawvere, F. W. & Schanuel, S. H. (1997). *Conceptual Mathematics*, Cambridge: Cambridge University Press.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*, Napoli: Bibliopolis.

Plotkin, G. D. (1983). *Domains*. (Available from http://homepages.inf.ed.ac.uk/gdp/publications/).

Robinson, J. A. (1965). 'A macine-oriented logic based on the resolution principle,' *Journal of the Association for Computing Machinery*, *12*, 23–41.

Stoy, J. E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Massachusetts: The MIT Press.

高橋正子 (2003). 『計算論 —— 計算可能性とラムダ計算』, 第 24 巻, コンピュータサイエンス大学講座, 近代科学社.

Troelstra, A. S. & Schwichtenberg, H. (2000). *Basic Proof Theory*, *43* of Cambridge Tracts in Theoretical Computer Science, Cambridge: Cambridge University Press, 2nd edition.

横内寛文 (1994). 『プログラム意味論』, 第 7 巻, 情報数学講座, 共立出版.

〔龍谷大学非常勤講師・哲学〕